```
// LEDs - maybe declare subsets and allocate each FPGA some
// great care has to be taken if both FPGAs try to access the same LEDs
/////////////////////////////////////////////////
macro expr LED_pins = {data = {     "AU13", "AT14", "AV12", "AU14",
                                                        "AW12", "AT15", "AV13",
"AU15"}};



/////////////////////////////////////////////////////
// ATA Interface
/////////////////////////////////////////////////////
macro expr ATA_pins = {data = {     "AU26", "AV27", "AT26", "AW28", "AU27",
                                                        "AV28", "AW29", "AT27",
"AW30", "AU28",
                                                        "AV30", "AV29", "AW31",
"AU29", "AV31",
                                                        "AT29", "AW32", "AU30",
"AW33", "AT30",
                                                        "AV33", "AU31", "AT31",
"AW34", "AV32",
                                                        "AV34", "AU32", "AW35",
"AT32", "AV35",
                                                        "AU33", "AW36",
"AT33"}};
```

```
///////////////////////////////////////////////////////
// Expansion Bus (32 bits)
/////////////////////////////////////////////////////////

macro expr E_pins = {data = {        "AV17", "AU18", "AW17", "AT19", "AV18",
                                        "AU19", "AW18", "AU21",
"AV19", "AW20",
                                        "AV20", "AR22", "AV23",
"AW21", "AU23",
                                        "AV21", "AT23", "AW22",
"AR23", "AV22",
                                        "AV24", "AW23",
"AW24", "AU24", "AW25",
                                        "AT24", "AV25", "AU25",
"AW26", "AT25",
                                        "AV26", "AW27"}};


/////////////////////////////
// Serial H Bus
////////////////////////////
macro expr SERIALH_pins = {data = {"F39", "H37", "F38", "H36", "E39", "G37",
"E38"}};



/////////////////////////////////////////
```

```
// SelectLink Bus - Directly connects the 2 FPGAs
////////////////////////////////////////

macro expr SL_pins = {data = {       "AV3", "AU4", "AV5",  "AT6", "AV4",  "AU6",
                                                          "AW4", "AT7", "AW5",
"AU7", "AV6",  "AT8",
                                                          "AW6", "AU8", "AV7",
"AT9", "AW7",  "AV8",
                                                          "AU9", "AW8", "AT10",
"AV9", "AU10", "AW9",
                                                          "AT11","AV10","AU11",
"AW10","AU12", "AV11",
                                                          "AT13", "AW11"}};




//////////////////////////
//VGA interface
//////////////////////////
macro expr VGA_pins = {data = {    "AW13", "AV14", "AT16", "AW14", "AU16",
                                                          "AV15", "AR17", "AW15",
"AT17", "AU17",
                                                          "AV16", "AR18", "AW16",
"AT18"}};

macro expr vga_vsync_pin = { data = { "AV14" } };
macro expr vga_hsync_pin = { data = { "AW13" } };
macro expr vga_data_pins = { data = {   "AT16", "AW14", "AU16", "AV15",
                                  "AR17", "AW15", "AT17", "AU17",
                                  "AV16", "AR18", "AW16", "AT18"} };


// macros for compatibility with existing programs
macro expr vsync_pin = { "AV14"  };
macro expr hsync_pin = { "AW13" };
macro expr video_spec = { data = {  "AT16", "AW14", "AU16", "AV15",
                              "AR17", "AW15", "AT17", "AU17",
                              "AV16", "AR18", "AW16", "AT18"} };


//////////////////////////
// CPLD interface pins
//////////////////////////

macro expr BUSMaster_pin = { data = { "C26" }}; // P12
macro expr FPcom_pins = { data = { "B26", "C27", "A27"}}; //P14 P15 P16

//////////////////////////
// Serial Ports pins
//////////////////////////


macro expr SERIAL_pins = {data = {"AV36", "AU34", "AU36", "AT34"}};


macro expr rs232_txd_pin = {data = { "AV36"}};
macro expr rs232_rxd_pin = {data = { "AU36"}};
macro expr rs232_rts_pin = {data = { "AU34"}};
macro expr rs232_cts_pin = {data = { "AT34"}};



////////////////////////////////////////////////////////////////////////
// USB
////////////////////////////////////////////////////////////////////////

macro expr USBMaster_pin = { data = { "D26" }}; // P13

macro expr USBD_pins = {data = {"C29", "A30", "D29", "B30", "C30", "A31", "D30",
"A32"}};

macro expr USBMS_pins = { data = {"D27"} };

macro expr USBnRST_pins = { data = {"B27"} };

macro expr USBIRQ_pins = { data = {"C28"} };

macro expr USBA0_pins = { data = {"A28"} };

macro expr USBnRD_pins = { data = {"B28"} };

macro expr USBnWR_pins = { data = {"B29"} };

macro expr USBnCS_pins = { data = {"A29"} };
```

```
#endif _KOMPRESSOR_SLAVE_HEADER
```

Appendix C

Following is a description of a parallel port interface that gives full access to the all the parallel port pins and implements a parallel port data transfer functionality that can be used in conjuction with the ESL download utility

```
// *************************************************************
// Parallel port controller
// *************************************************************

// Instantiates a component that controls the parallel port.
// This is to be run in parallel in the main loop. The interfaces
// provide the user with abstracts to use deal efficiently with the
// component.

// ****************************************
// Interfaces
//
// API to Parallel Port - for direct access to the pins
//
// PpWriteData((unsigned 8)byte) -- write byte to data pins
// PpReadData((unsigned 8)byte) -- read byte from data pins
// PpReadControl((unsigned 4)control_port) -- read the control port
// PpReadStatus((unsigned 6)status_port) -- read the status port
// PpSetStatus((unsigned 6) status_port) -- write to the status port
//
//
// API for the ESL parallel data transfer utility
//
```

```
// OpenPP(error) -- open the parallel port for data transfer
// ClosePP(error) -- close the port
// SetSendMode(error) -- set the port to send mode
// SetRecvMode(error) -- set the port to receive mode
// SendPP(byte, error) -- send a byte over the port
// ReadPP(byte, error) -- read a byte from the port
//
// error returns the result of the command:
// 0 - no error
// 1 - buffer error
// 2 - timeout error
//
// Note: SendPP and ReadPP will block the thread until  a byte is transmitted or the
timeout
// value is reached. If you need to do some processing while waiting for a
communication
// use a 'prialt' statement to read from the global pp_recv_chan channel or write to the
// pp_send_chan channel.


////////////////////////////////////////////////////////////////////////
//  The Nitty Gritty
////////////////////////////////////////////////////////////////////////


// The necessary channels
chan unsigned 8 pp_send_chan, pp_recv_chan;
chan unsigned 2 pp_command, pp_error;
chan pp_data_send_channel, pp_data_read_channel, pp_control_port_read;
chan pp_status_port_read, pp_status_port_write;


#define OPEN_CHANNEL   0
#define CLOSE_CHANNEL 1
#define SEND_MODE              2
#defineRECV_MODE              3

#define PP_NO_ERROR             0
#define PP_HOST_BUFFER_NOT_FINISHED      1
#define PP_OPEN_TIMEOUT 2

// Currently the functions don't act on any errors, but this can easily be added if
required.
// return of error code could also be used to generate a time-out condition.


macro proc OpenPP(error)
{
        pp_command ! OPEN_CHANNEL;
        pp_error ? error;
}


macro proc ClosePP(error)
{
        pp_command ! CLOSE_CHANNEL;
        pp_error ? error;
}

macro proc SetSendMode(error)
{
        pp_command ! SEND_MODE;
        pp_error ? error;
}

macro proc SetRecvMode(error)
{
        pp_command ! RECV_MODE;
        pp_error ? error;
}


macro proc WritePP(byte, error)
{
        pp_send_chan ! byte;
}


macro proc ReadPP(byte, error)
{
        pp_recv_chan ? byte;
}


// ************************************************************
// Parallel port controller
// ***********************************************************

// Host Channel Control (HCC)   nAutoFeed
// FPGA Channel Control (FCC)   DONE
// Host Data Control (HDC)        nSelect_in
// FPGA Data Control (FDC)        nACK
// FPGA ready to communicate (FRTC)  PE


// HCC indicates that host is sending -  end of the buffer
// FCC controls direction of commmunication
// FRTC indicates that FPGA is ready
// when FPGA sets FCC low, rising edge on FDC when data applied
// lower when host responds with HDC high
// when FCC high FPGA is in receive mode and host applies data
// on rising edge on HDC. FPGA responds with FDC high and host
// then lowers HDC. Host will keep data byte on pins till FDC is
// lowered again by the FPGA

// chan unsigned 8 pp_data_chan;
// chan unsigned 4 pp_control_chan;
// chan unsigned 5 pp_status_chan;


////////////////////////////////////////////////
// Macro to implement ESLs bi-directional host-fpga
// data transfer protocol
// Accesses the physical layer
/////////////////////////////////////////////


macro proc Test_PP()
{

        unsigned 4 control_port;
        unsigned 6 status_port;

        unsigned 21 counter;

//      PpSetControl(0b0000);
        PpSetStatus(0b000000);

        do
        {
        counter++;
        }while(counter != 0);

        PpSetStatus(0b000001);

        do
        {
        counter++;
        }while(counter != 0);

        PpSetStatus(0b000010);
```

```
do
{
counter++;
}while(counter != 0);


PpSetStatus(0b000100);

do
{
counter++;
}while(counter != 0);


PpSetStatus(0b001000);

do
{
counter++;
}while(counter != 0);


PpSetStatus(0b010000);

do
{
counter++;
}while(counter != 0);
```

```
        PpSetStatus(0b000000);

                do
        {
        counter++;
        }while(counter != 0);

        PpSetStatus(0b011111);

        while(1)
        {
                PpReadControl(debug_control);
        }
}




macro proc pp_coms(pp_send_chan, pp_recv_chan, pp_command, pp_error)
{

        // bit masks for accessing control and status ports

//control_port = nSelect_in.in @ init.in @ nAutofeed.in @ nStrobe.in;
#define HCC control_port[1] //0b0010   //nAutofeed pin on control port
```

```
#define HDC control_port[2] //0b0100   //nInit pin on control port

//status_port = ppdir @ busy @ nAck @ pe @ select @ nError;
#defineFRTC 0b000010              //pe pin on status port
#define FCC    0b000100           //select pin on status port
#define FDC    0b001000           //nAck pin on status

#definePP_SEND 0b100000
#define PP_READ 0b000000


        unsigned 4 control_port;
        unsigned 6 status_port;
        unsigned 1 pp_dir with {warn = 0};
        unsigned 2 command;
        unsigned 8 temp_data;

        PpSetStatus(PP_READ | FRTC); // initialise the port, read mode, FRTC high

        while(1)
        {
                prialt
                {
                                case pp_command ? command:

                                        // deal with any commands received
                                        switch (command)
                                        {
                                        case OPEN_CHANNEL:
```

```
                                        // open channel and set to FPGA send
mode
                                        PpSetStatus(PP_SEND | FCC ); // |FDC
keep FCC low, FRTC low to indicate ready
                                        pp_dir = 1;

                                        // wait for pulse on HCC in response to
open channel                 .
                                        PpReadControl(control_port);

                                        while(HCC) // wait for nHCC to go low
                                        {
                                                PpReadControl(control_port);
                                        }

                                        while(!HCC) // wait for nHCC to go high
                                        {
                                                PpReadControl(control_port);
                                        }
```

```
        pp_error ! PP_NO_ERROR;

        break;



    case CLOSE_CHANNEL:  // closes the channel
regardless of state
        PpSetStatus(PP_READ  | FRTC); // sets
status port to all zeros, FRTC high
        pp_dir = 0;
        pp_error ! PP_NO_ERROR;
        break;



    case SEND_MODE:

        PpReadControl(control_port);


        // set FRTC  high - host send, start driving
data pins, FCC low
        PpSetStatus(PP_SEND);
        pp_dir = 1;
        pp_error ! PP_NO_ERROR;

        // BUFFERNOTFINISHED
        break;
```

```
case RECV_MODE:

        // set FRTC  high - host read - stop driving
data pins, FCC high, FDC low
        PpSetStatus(PP_READ | FCC );
//|FDC|FCC
        pp_dir = 0;
        pp_error ! PP_NO_ERROR ;

        break;


default:
        delay;
        break;
}

break;



// FPGA sending
case pp_send_chan ? temp_data:


        PpSetStatus(PP_SEND); // FCC low, FDC
low - pin is inverted
```

```
PpReadControl(control_port);

while(!HCC) // wait for host to de-assert HCC
{
        PpReadControl(control_port);
}

PpWriteData(temp_data);
PpSetStatus(PP_SEND | FDC);// FCC low, FDC high

PpReadControl(control_port);

while(!HDC) // wait for host to assert HDC
{
        PpReadControl(control_port);
}

PpSetStatus(PP_SEND); // FCC low, FDC low - pin is inverted

PpReadControl(control_port);

while(HDC) // wait for host to de-assert HDC
```

```
                    {
                            PpReadControl(control_port);
                    }

                    break;

                    // host sending
            default:

                    PpReadControl(control_port);
                    PpReadStatus(status_port);

                    if (!status_port[5] & !HCC) // read one
byte, if in read mode and HCC is low
                    {

                            while(!HDC)  // wait for host to
apply data and raise HDC
                            {

        PpReadControl(control_port);

                            }
```

```
                                        PpSetStatus( PP_READ | FCC |
FDC); // FCC high FDC high

                                        PpReadData(temp_data);

                                        pp_recv_chan ! temp_data;

                                        PpReadControl(control_port);
                                        PpReadStatus(status_port);

                                        while(HDC) // wait for host to
remove HDC
                                        {
        PpReadControl(control_port);
                                        }

                                        PpSetStatus( PP_READ | FCC ); //
FCC high FDC low

                                    }
                                    else delay;

                                    break;

                }

            } // while(1)
```

```
                delay; // avoid combinational cycles
}




////////////////////////////////////////////////////////
// Parallel Port - Physical layer
//
// Allows access to all the data, control and status ports
// through a series of channels which can be read from
// and written to.
////////////////////////////////////////////////////////

// Macro abstractions for the various actions

macro proc PpWriteData(/*(unsigned 8)*/ byte)
{
        pp_data_send_channel ! byte;

}


macro proc PpReadData(/*(unsigned 8)*/ byte)
{
        pp_data_read_channel ? byte;
```

```
}

macro proc PpReadControl(/*(unsigned 4)*/ control_port)
{
        pp_control_port_read ? control_port;

}


macro proc PpReadStatus(/*(unsigned 6)*/ status_port)
{
        pp_status_port_read ? status_port;

}

macro proc PpSetStatus(/*(unsigned 6)*/ status_port)
{
        pp_status_port_write ! status_port;
}




// Actual Parallel Port control circuitry

macro proc parallel_port(pp_data_send_channel, pp_data_read_channel,
pp_control_port_read,
                                                pp_status_port_read,
pp_status_port_write)
{
        unsigned 8 pp_data;
        unsigned 6 status_register;

        interface  bus_ts_clock_in (unsigned 8) data_bus(pp_data, status_register[5])
with pp_data_pins;


        // Control Port (unsigned 4, made up as nSelect_in.in @ init.in @ nAutofeed.in
@ nStrobe.in)
        interface bus_clock_in (unsigned 4) control_port() with control_port_pins;


        //  Status Port, status_register = pp_direction @ busy @ nAck @ pe @ Select @
nError;
        interface bus_out() status_port_bus(status_register[4:0]) with status_port_pins;

        // Setting pp_direction to 1 will drive data onto the pins.

        while(1)
        {
                // Allows read of control, read / write of status and data ports
simulatneously
                par
                {
```

```
prialt
{
        case pp_control_port_read ! control_port.in:
                break;

        default:
                delay;
                break;
}


prialt
{
        case pp_status_port_write ? status_register:
                break;

        case pp_status_port_read ! status_register:
                break;

        default:
                delay;
                break;
}


prialt
{
        case pp_data_send_channel ? pp_data:
```

```
                                        break;

                                case pp_data_read_channel ! data_bus.in:
                                        break;

                                default:
                                        delay;
                                        break;
                        }


                }

        }
        delay; // to avoid combinational cycles
}




        //macro expr control_port = nSelect_in.in @ init.in @ nAutofeed.in @
nStrobe.in;

        /*interface  bus_clock_in (unsigned 1) nAutofeed() with nAutoFeed_pin;
        interface  bus_clock_in (unsigned 1) init() with init_pin;
        interface  bus_clock_in (unsigned 1) nSelect_in() with nSelect_in_pin;
        interface  bus_clock_in (unsigned 1) nStrobe() with nStrobe_pin;

        // defined in the same order as on a PC
        macro expr control_port = nSelect_in.in @ init.in @ nAutofeed.in @ nStrobe.in;
*/


        /*
        interface  bus_out () nAck_line( status_register[3] ) with nAck_pin;
        interface  bus_out () busy_line(status_register[4]) with busy_pin;
        interface  bus_out () pe_line(status_register[2]) with pe_pin;
        interface  bus_out () select_line(status_register[1]) with select_pin;
        interface  bus_out () nError_line(status_register[0]) with nError_pin;
        */


        // status_register[5] is high to send and low to receive
        // defined in the same order as on a PC
//      macro expr status_port = pp_direction @ busy @ nAck @ pe @ Select @
nError;
```

Appendix D

This Appendix describes a Macro Library for a board according to the present invention. The library contains functions for

1) Memory arbitration
2) Flash bus arbitration
3) Read and Write to Flash RAM
4) FPCOM settings
5) Control of the LEDs

```
////////////////////////////////////////////////////////////
//
// Interfaces
//
// Shared RAM arbitration
//        ----------------------
//        KRequestMemoryBank(bankMask)
//        KReleaseMemoryBank(bankMask)
//
//        Flash RAM Macros
// ----------------
//        KEnableFlash()
//        KDisableFlash()
//        KSetFlashAddress(address)
//        KWriteFlashData(address, data)
//        KReadFlashData(address, data)
//        KReadFlashID(flash_component_ID, manufacturer_ID)
//
//
```

```
//        Flash bus arbitration
//        ---------------------
//        KSetFPGAFBM()
//        KReleaseFPGAFBM()
//
//        Others
//        ---------
//        KSetLEDs(maskByte)
// KSetFPCOM(fpcom)


#ifndef _KOMPRESSOR_LIBRARY
#define _KOMPRESSOR_LIBRARY

// Include header file
//#include "KompressorMaster.h"


/////////////////////////////////////////////////////////////////////////
// Request access to a memory bank
//
// The procedureS will block until access to all the requested banks have been
// granted.
//
```

```
unsigned 1 shared_bank0_request = 1 with { warn = 0} ;
unsigned 1 shared_bank1_request = 1 with { warn = 0} ;

interface bus_out() shbk0req(shared_bank0_request) with
sram_shared_bank0_request_pin;
interface bus_out() shbk1req(shared_bank1_request) with
sram_shared_bank1_request_pin;
interface bus_clock_in(unsigned 1) shbk0grant() with sram_shared_bank0_grant_pin;
interface bus_clock_in(unsigned 1) shbk1grant() with sram_shared_bank1_grant_pin;



macro proc KRequestMemoryBank0()
{
        shared_bank0_request = 0;
        while(shbk0grant.in) delay;
}



macro proc KRequestMemoryBank1()
{
        shared_bank1_request = 0;
        while(shbk1grant.in) delay;
}
```

```
//////////////////////////////
// Release a memory bank
//


macro proc KReleaseMemoryBank0()
{
        shared_bank0_request = 1;
}


macro proc KReleaseMemoryBank1()
{
        shared_bank1_request = 1;
}





//////////////////////////////////////////////
//
// Functions for dealing with FP commands

#define FP_SET_IDLE              (unsigned 3)   7
#define FP_READ_STATUS (unsigned 3)   5
#define FP_CCLK_LOW              (unsigned 3)   3
```

```
#define FP_CCLK_HIGH      (unsigned 3)   7
#define FP_WRITE_CONTROL (unsigned 3)        0



unsigned 3 fpcom = FP_SET_IDLE  with { warn = 0}; // default
interface bus_out() fpcom_bus(fpcom) with FPcom_pins;

macro proc KSetFPCOM(command)
{
        fpcom = command;
        delay;
        delay;
}


macro proc KReadCPLDStatus(status)
{
  par
        {
  KDisableFlash();
        flash_write = 0;
        }

  KSetFPCOM(FP_READ_STATUS);

        delay;
        delay;
        delay;
  delay;
```

```
    status = flash_data_bus.in;

    par
        {
            KSetFPCOM(FP_SET_IDLE);
            KEnableFlash();
        }
}


macro proc KWriteCPLDControl(control)
{
    KDisableFlash();
    par
    {
        flash_data  = (unsigned 8) (0 @ control);
        flash_write = 1;
    }

    KSetFPCOM(FP_WRITE_CONTROL);
    delay;
    delay;
    delay;
    par
    {
        KSetFPCOM(FP_SET_IDLE);
        flash_write = 0;
        KEnableFlash();
```

```
          }
}


//////////////////////////////////
//
//        Flash RAM stuff
//
//
// Parameters;
//
//        Read/write cycle            120ns
//        Address to output           120ns
//        CE to ouput                           120ns
//
//        CE low to WE low            0
//        write pulse width low 70ns
//        data setup to we high  50ns
//        address setup to we hi 55ns
//        address/data hold           0ns
//        write pulse width high30ns



unsigned 24 flash_address  with { warn = 0};
unsigned 8 flash_data  with { warn = 0};
unsigned 1 flash_cs = 1, flash_we = 1, flash_oe = 1  with { warn = 0}; // initialise to
high
```

```
unsigned 1 flash_write = 0 with { warn = 0}; // controls direction of the data pins
unsigned 1 flash_on = 0 with { warn = 0}; // controls the other tristate buses

interface bus_ts_clock_in(unsigned 24) flash_address_bus(flash_address, flash_on)
with {data = FA_pins};
interface bus_ts_clock_in(unsigned 1) flash_chipselect(flash_cs, flash_on) with
flash_cs_pin;
interface bus_ts_clock_in(unsigned 1) flash_writeenable(flash_we, flash_on) with
flash_we_pin;
interface bus_ts_clock_in(unsigned 1) flash_outputenable(flash_oe, flash_on) with
flash_oe_pin;
interface bus_ts_clock_in(unsigned 8) flash_data_bus(flash_data, flash_write) with
{data = FD_pins};


macro proc KEnableFlash()
{
        par
        {
        flash_on = 1;
        flash_cs = 0;
        }
}


macro proc KDisableFlash()
{
        par{
        flash_on = 0;
        flash_cs = 1;
        }
}



// Sets up the address on the
macro proc KSetFlashAddress(address)
{
        flash_address = address;
}



macro proc KWriteFlashData(address, data)
{

        par // set up address and data and drive onto pins
        {
        flash_oe = 1; // disable output
        flash_address = address;
        flash_data = data;
        flash_write = 1;
        flash_we = 0;  // send write pulse
        }

        // running at 50/2 MHz - 40 ns cycles - 2 delays should be
        // sufficient to meet timing constraint

        delay;
```

```
  delay;

      par
      {
            flash_we = 1;
            flash_write = 1;
      }

}

macro proc KReadFlashData(address, data)
{
      par
      {
      flash_write = 0;
      flash_oe = 0; // enable output
      flash_address = address;
      }

      // running at 50/2 MHz - 40 ns cycles - 2 delays should be
      // sufficient to meet timing constraint
      delay;
  delay;
      data = flash_data_bus.in;

}


macro proc KReadFlashID(flashid, manid)
```

```
{

    par
    {
        KEnableFlash();
        KSetFPGAFBM();
    }

    KWriteFlashData(0, 0x90);
    KReadFlashData(0, manid);
    KReadFlashData(2, flashid);

    par
    {
    KReleaseFPGAFBM();
    KDisableFlash();
    }

}


macro proc KReadFlashStatus(status)
{
        par
        {
            KEnableFlash();
            KSetFPGAFBM();
        }
```

```
                KWriteFlashData(0, 0x70);
                KReadFlashData(0, status);

                par
                {
                        KDisableFlash();
                        KReleaseFPGAFBM();
                }

}


////////////////////////////////////
// Flash bus arbitration pins
//
unsigned 1 fbus_master = 1  with {warn = 0}; // initialise to not master
interface bus_out() bus_master_line(fbus_master) with BUSMaster_pin;

macro proc KSetFPGAFBM()
{
        fbus_master = 0;
}


macro proc KReleaseFPGAFBM()
{
        fbus_master = 1;
}
```

```
/////////////////////////////////////////////////////
// LED control macros


unsigned 8 LED = 0  with {warn = 0}; // by default
unsigned 1 LED_en = 0 with {warn = 0};
interface bus_ts(unsigned 8) LEDpins(LED, LED_en) with LED_pins;
macro proc KSetLEDs(maskByte)
{
   par
   {
         LED = maskByte;
   LED_en = 1;
   }
}



///////////////////////////////////
//
// FPcom ==7 CCLK = High
//
// From the FPGA BUSMuster pin should be brought low and the FLASH may be
// accessed as any normal device RAM device.
//
#endif _KOMPRESSOR_LIBRARY
```